5 CONCLUSIONS AND FUTURE WORK

The experience gave me enough of the mathematical viewpoint that it enabled me to find my way in the future, to be a rather theoretical computer scientist and yet still worry about the engineering consequences of what I was doing.

— Jeffrey D. Ullman

Owing to the growing adoption of WebAssembly, we have focused on the security of WebAssembly programs and the potential usages of Software Diversification. This thesis introduces a comprehensive set of methods and tools for Software Diversification in WebAssembly. It includes the technical contributions of this dissertation: CROW, MEWE, and WASM-MUTATE. Additionally, we present specific use cases for exploiting the diversification created for WebAssembly programs, along with their corresponding experimental results. In this chapter, we initially summarize the technical contributions of this dissertation. We then offer a succinct overview of the empirical findings of our research. Lastly, we delineate challenges and future research directions in this field.

5.1 Summary of technical contributions

CROW is a strategy that utilizes the LLVM compiler and requires the source code or the LLVM IR representation for its functioning. Its core comprises an enumerative synthesis implementation. Employing SMT solvers for functionality verification, CROW ensures the functional equivalence of the generated variants.

MEWE, on the other hand, enhances CROW by using identical core technology to generate program variants. Furthermore, it encapsulates the LLVM IR variants into a WebAssembly multivariant, thereby facilitating MVE through execution path randomization. Both CROW and MEWE are fully automated systems, necessitating only the input source code from users.

[0] Compilation probe time 2023/10/27 12:21:33

WASM-MUTATE, a tool based on binary, uses a set of rewriting rules and the input Wasm binary to generate program variants. The generation of WebAssembly variants primarily involves random e-graph traversals. Importantly, WASM-MUTATE eliminates the need for compiler adjustments, thus ensuring compatibility with all existing WebAssembly binaries. Unlike CROW and MEWE, which are confined to code and function sections, WASM-MUTATE can generate variants by transforming any segment of the Wasm binary.

CROW, MEWE, and WASM-MUTATE are open-source, public tools, making their deployment entirely practical. Notably, WASM-MUTATE is currently in use in real-world scenarios to enhance WebAssembly compiler tests[1].

5.2 Summary of empirical findings

According to the comparison of our technical contributions discussed in Chapter 3 and, the results of our use case experiments in Chapter 4 we summarize the following empirical findings.

Implications of our implementations: CROW and MEWE depend on SMT solvers to prove functional equivalence in their enumerative synthesis implementation, which can be a bottleneck in variant generation. Consequently, WASM-MUTATE produces more unique variants than CROW and MEWE by at least an order of magnitude. The main reason is that WASM-MUTATE uses a preset of rewriting rules accompanied by virtually inexpensive random e-graph traversals. Yet, by employing enumerative synthesis, CROW and MEWE ensure that the generated variants are more effectively preserved. The applications of our technical contributions are not orthogonal but complementary. Specifically, one can employ CROW and MEWE to generate a set of variants, which subsequently serve as rewriting rules for WASM-MUTATE. Furthermore, when practitioners require swift generation of variants, they could utilize WASM-MUTATE, accepting a decrease in preservation of the variants.

Offensive Software Diversification: We employ WASM-MUTATE to demonstrate the practicality of introducing Offensive Software Diversification to WebAssembly. Our research verifies the existence of opportunities for the malware detection community to bolster the automatic detection of cryptojacking WebAssembly malware. The results of our study are not only actionable, but they also provide measurable evidence of specific malware transformations that detection methods can target. One potential contributing factor to the success of WASM-MUTATE's evasion is a false sense of resilience. Prior research into the static detection of WebAssembly malware has exposed a flawed presumption that obfuscation techniques for WebAssembly are absent [**? ? ? ? ?**], whilst our

[1] `https://github.com/bytecodealliance/wasm-tools`

software diversification tools present a viable solution for enhancing the precision of WebAssembly malware detection systems.

Defensive Software Diversification: Our techniques enhance overall security by facilitating the deployment of unique and diversified WebAssembly binaries, potentially utilizing different variants as needed. For instance, WASM-MUTATE generates Wasm binaries that are resistant to Spectre-like attacks. Given that WASM-MUTATE can generate tens of thousands of variants within minutes, it becomes feasible to deploy a unique variant for each function invocation on FaaS platforms. This rationale applies equally to both CROW and MEWE. Our tools can mitigate other side-channel attacks. CROW, for example, excels at creating Wasm variants that reduce side-channel noise, thereby strengthening defenses against potential side-channel attacks. Another approach is the deployment of multivariants from MEWE, which can impede high-level timing-based side-channels [**?**].

TODO TBD: Memory obliviousness does not means that the memory patterns of the execution does not change. At least for Wasm.

5.3 Future Work

Along with this dissertation we have highlighted several open challenges related to Software Diversification in WebAssembly. These challenges open up several directions for future research. In the following, we outline some of these directions.

Extending WASM-MUTATE: WASM-MUTATE may gain advantages from the enumerative synthesis techniques employed by CROW and MEWE. Specifically, WASM-MUTATE could adopt the transformations generated by these tools as rewriting rules. This approach could enhance WASM-MUTATE in two specific ways. Firstly, it could improve the preservation of the variants generated by WASM-MUTATE. The primary reason for this is that enumerative synthesis outperforms compiler optimizations, making it more challenging for reverse engineers to identify the original program. Secondly, this method would inevitably expand the diversification space of WASM-MUTATE e-graphs.

Program Normalization: We successfully employed WASM-MUTATE for the evasion of malware detection (see Section 4.1). The proposed mitigation in the prior study involved code normalization as a means of reducing the spectrum of malware variants. Our current work provides insights into the potential effectiveness of this approach. Specifically, a practically costless process of pre-compiling Wasm binaries could be employed as a preparatory measure for malware classifiers. In other words, a Wasm binary can first be JITed to machine code, effectively eliminating approx. 24% of malware variants according to our preservation statistics highlighted in Table 3.1. This approach could substantially enhance the efficiency and precision of malware detection systems.

Meta-oracles: Our experiment results in Section 4.1 indicate that VirusTotal surpasses MINOS in detecting WebAssembly cryptojacking. The primary factor contributing to this is VirusTotal's utilization of a broader range of antivirus vendors, which employs various detection strategies. On the other hand, MINOS functions as a binary oracle. This evidence supports the use of multiple malware oracles (meta-oracles) in identifying cryptojacking malware in browsers. In the context of WebAssembly, given the existence of numerous and diverse Wasm-specific detection mechanisms, this strategy is both practical and feasible, yet not explored in the literature.

Fuzzing WebAssembly consumers: Generating well-formed inputs in fuzzing campaigns presents a significant challenge [**?**]. This challenge is particularly prevalent in fuzzing compilers, where the inputs need to be executable yet complex enough programs to test various compiler components. To address this issue, our tools can generate semantically equivalent variants from an original Wasm binary, thereby improving the scope and efficiency of the fuzzing process. A practical instance of the effectiveness of this approach occurred in 2021, when it led to the discovery of a security CVE in wasmtime [**?**].

Mitigating Port Contention: Rokicki et al. [**?**] demonstrated the feasibility of a covert side-channel attack using port contention within WebAssembly code in the browser. This attack essentially depends on the precise prediction of Wasm instructions that trigger port contention. To address this security issue, one could conveniently implement Software Diversification as a browser plugin. Software Diversification has the capability to substitute the Wasm instructions used as port contention predictors with other instructions. This is very similar to the effect on timers and padding previously discussed in Section 4.2. Such an approach would inevitably eliminate the port contention in the specific port used for the attack, thereby strengthening browsers against such harmful actions.

AI and Software Diversification: As discussed in Chapter 3, implementing a diversifier at the high language level seems impractical due to the multitude of existing frontends. However, the emergence of Large Language Models (LLMs) and their ability to generate high-level language may address this problem. Nevertheless, we argue that simply connecting the LLM to the diversifier does not provide a complete solution; studies on preservation must also be conducted. Specifically, high-level diversification might lead to low preservation, thereby challenging the assumption of diversification at the low-level. In the context of WebAssembly, considering the wide variety of frontends, utilizing LLMs might be a feasible method for generating Software Diversification. Although preservation poses a problem at a high-level, it could potentially solve the inherited, more challenging issue of transforming programs at the intermediate representation level or WebAssembly bytecode itself.